



資訊管理學系 陳士杰老師

# 作業系統

## Operating Systems

### 執行緒

### Thread



國立聯合大學  
NATIONAL UNITED UNIVERSITY



## ■ 本章重點

- ✚ Thread定義
- ✚ 與Process的比較
- ✚ User Level v.s. Kernel Level Thread
- ✚ Models種類
- ✚ Thread Pool





# Thread (執行緒)

## Def:

- 又稱為 **Lightweight Process (LWP; 輕量級行程)**, 是OS分配CPU Time的對象單位。
- 和Process相似, 也是搶奪CPU的一個基本單位。

## 每個Thread擁有下列項目:

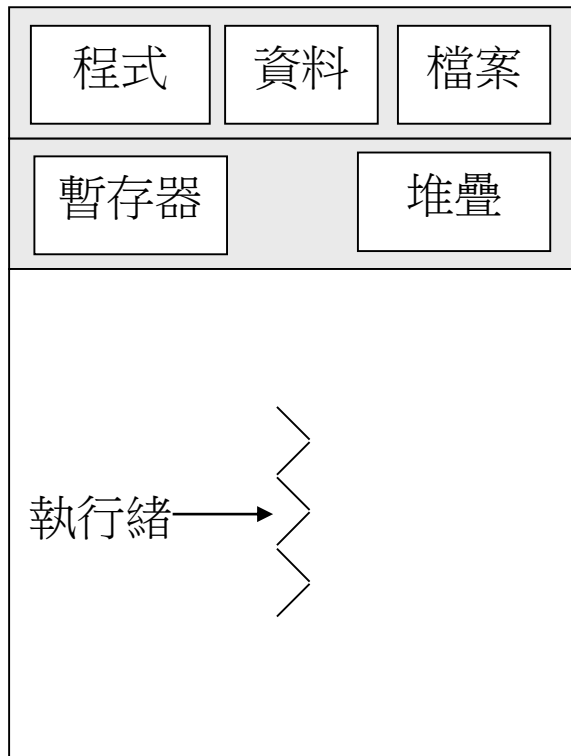
- Thread ID (執行緒的識別碼)**
- Thread State**
- Program counter (程式計數器)**
- Register set (暫存器組)**
- Stack (堆疊)**

## 同一個Task (Process)內的Threads彼此共享:

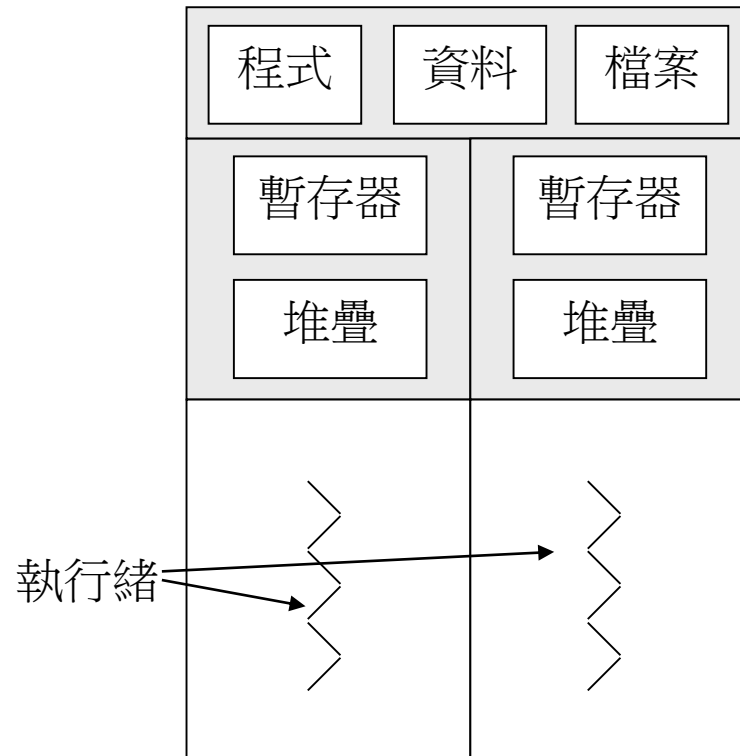
- Code Section (程式碼區域)**
- Data Section (資料區域)**
- O.S. Resources (作業系統資源)**

## 傳統的Process (Heavyweight Process; 重量級行程)就等同於一個Task (Process)內**只有單一個Thread**。





單執行緒  
(即：Process)



多執行緒

✿ **Process和Thread只有內部OS較能看得出來，外觀看不出來!!**

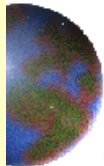




# Thread vs. Process

Thread	Process
1. Light Weight Process	1. Heavy Weight Process
2. 同一個Task (或Process)內的Threads可以共享Code Section, Data Section, 及O.S. Resources	2. 不同的Process之間無共享的Address Space, 互為獨立。
3. Context Switching 負擔輕	3. 負擔重
4. Thread的管理(Management)成本低 (管理項目: Creation, Scheduling, Context Switching...etc.)	4. 成本高
5. 一個Task內有多條Threads存在	5. 一個Task內只有一條Thread
6. Process (Task) 內的某Thread被Block, 則可切到其它Thread執行。此時, 若Process內只要還有Thread在執行, 則Process不會被Block	6. Process (Task) 內的單一Thread被Block, 則整個Process會被Block。
7. 能充份利用Multiprocessing架構的效益, 因為同一個Process的不同Threads可以同時在不同的CPU執行	7. 較無法發揮SMP之Parallel Processing效益
8. 必須提供對Share Data的互斥存取控制, 以防止不正常Thread存取所造成之危害	8. Process與Process無共享, 所以較無必要





# ■ Benefits (利益)

## ✚ Responsiveness (應答)

- ✚ 多執行緒在一個交談互動的應用中，允許**程式中的某一部份當被中斷或是執行得非常久**時，該程式仍然可以繼續執行。(一個**Process**內只要還有一個**Thread**還在Run，則該**Process**還可再執行)

## ✚ Resource Sharing (資源分享)

- ✚ 執行緒間共用著它們所屬行程的記憶體和資源

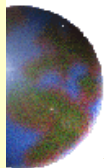
## ✚ Economy (經濟)

- ✚ 採用執行緒的話，因為**執行緒的產生**和從事**Context Switch**共用它們的所屬**Process**的記憶體和資源，所以在實行上比較經濟

## ✚ Utilization of MP Architectures (使用多處理器架構)

- ✚ 如果是在多處理器架構下實施多執行緒，因為每一個執行緒可以並行地在不同的處理器上執行，因此多執行緒的利益可以大幅提升。
- ✚ 在單一處理器的架構下，**CPU**讓每一個執行緒快速地切換移動，讓使用者誤以為是同時在進行。





# Thread 種類

- ❁ 分類角度 : **Thread Management** (e.g., Creation, Destroy, Context Switching, Scheduling, etc.) 是由誰去掌控？
- ❁ 執行緒可以依其執行時**所處模式**的不同，區分成兩個模式：
  - ❖ The user level -- **User Threads (使用者執行緒)**
    - 在User Mode下進行, O.S.不知道有這些Thread存在
    - 不需要O.S.介入管理
  - ❖ The kernel level -- **Kernel Threads (核心執行緒)**
    - 在Monitor Mode下進行, O.S.知道有這些Thread存在
    - 由O.S.介入管理

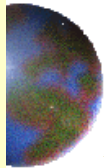




User Thread	Kernel Thread
1. <b>Def:</b> Thread Management是由使用者層次(User Level)的執行緒程式庫(Thread Library)所支援	1. <b>Def:</b> Thread Management是由Kernel所負責
2. Kernel不知道User Threads的存在	2. Kernel掌握所有Threads之管理
3. User Thread管理成本低	3. User Thread管理成本高
4. <b>優:</b> Thread之間的Scheduling, Context Switching不需Kernel介入。∴Fast Context Switching and Creation...	4. <b>缺:</b> Slower Context Switching and Creation...
5. <b>缺:</b> (1)當User Thread發出Blocking System Call, 且Kernel是Single Threaded, 會導致整個Process被Block住 (2)無法有效利用Multiprocessor效益	5. <b>優:</b> (1)Thread發出Blocking System Call不會使整個Process Blocked (2)可以安排不同Thread在不同CPUs上平行執行, 發揮Multiprocessor效益
6. POSIX: P-Threads Solaris 2: UI-Threads, Green-Threads Mach: C-Threads	6. Windows NT, 2000 Solaris 2 Tru 64 UNIX





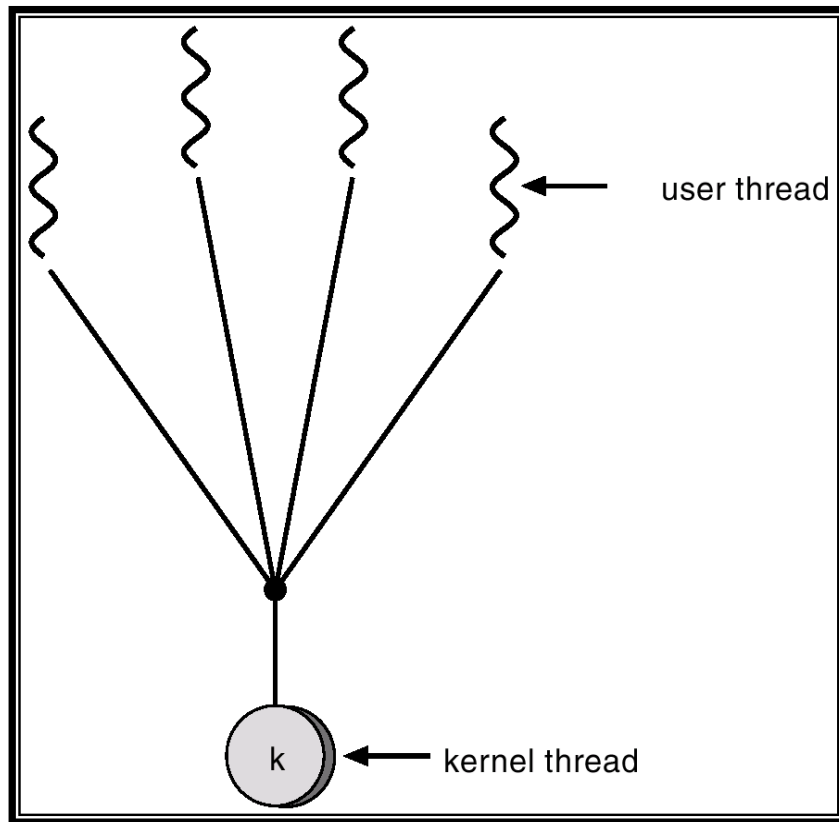


# ■ Multithreading Models (多執行緒模式)

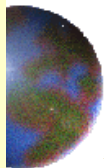
- ⊗ **Many-to-One (多對一模式)**
- ⊗ **One-to-One (一對一模式)**
- ⊗ **Many-to-Many (多對多模式)**



## Many-to-One Model (多對一模式)



- 多對一模式是指**多個User Threads**對應到**一個Kernel Thread**。
- 執行緒主要的管理動作是在**使用者空間**執行，所以很有效率。
- 缺點是如果有任何一個**User Level**的**Thread**執行暫停的系統呼叫，將**造成整個行程暫停執行**。
- 雖然這個模式可以產生它所需要的**Thread**數量，但因為**只有一個Kernel Thread**可以存取**Kernel**，**O.S.**一次只能使用一個執行緒，且**O.S.**不知道有這些**User Threads**存在，無法將**Block**以外的**Thread**配給其它的處理器，所以**數個執行緒不能在多個處理器上並行執行**。

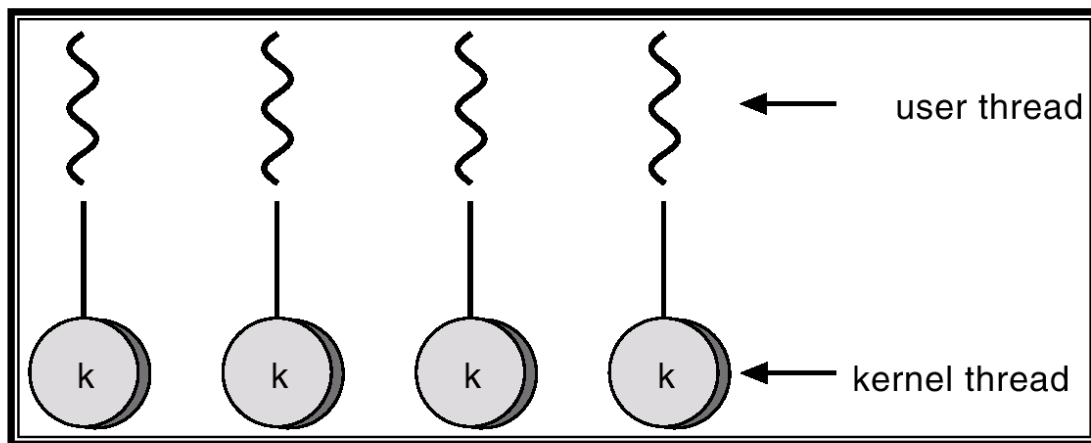


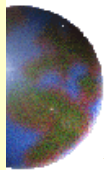
## One-to-One Model (一對一模式)

- ✚ 每一個**User Thread**都對應到一個**Kernel Thread**。當一個使用者執行緒處於暫停或是等待的狀態時，其它的執行緒都還可以執行。
- ✚ 它比多對一模式提供了更多的**並行功能**，也允許**多個Threads在Multiprocessor上並行執行**。
- ✚ 產生一個**User Thread**時，需連帶產生一個**Kernel Thread**，而**Kernel Thread**會對程式的執行產生一些額外的負擔。 $\therefore$ 此模式**限制執行緒產生的個數**。

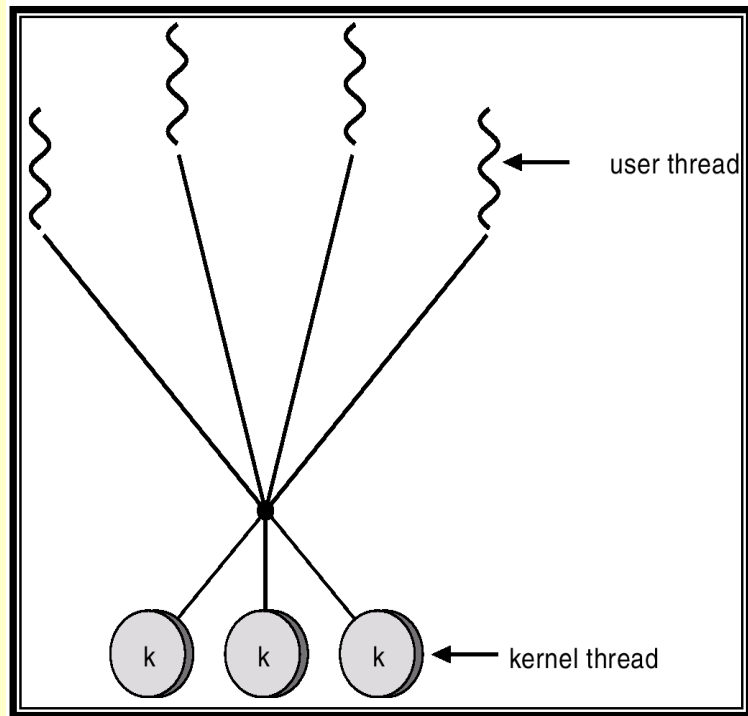
### ✚ Examples

- ❏ Windows NT/2000
- ❏ OS/2





## Many-to-Many Model (多對多模式)



✚ 多個**User Threads**對應到多個(個數少於或等於使用者執行緒個數)**Kernel Threads**。

- ✚ 多對一模式可以產生它所需要的執行緒數量，但因為核心一次只能使用一個執行緒，所以數個執行緒**不能真正地在多處理器的環境下並行執行**。
- ✚ 一對一模式雖然可以提供較強的並行能力，但**不能產生太多的執行緒**。

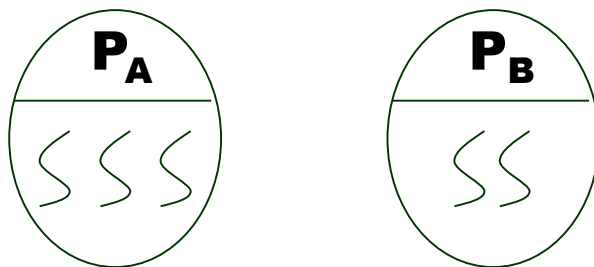
✚ **Programmer**或**O.S.**可以產生所需要的**Thread**數目，使其在多處理器上並行執行；另外，當一個**Thread**暫停執行時，**O.S.**可以安排另一個**Thread**接著執行。





## ❖ 範例 ❖

有兩個 **Process A** 與 **B**



若 **OS** 採平均分配原則來分配 **CPU Time**, 則 **P<sub>A</sub>** 與 **P<sub>B</sub>** 各分多少 % 之 **CPU Time**。1) **User Thread**, 2) **Kernel Thread**

**Ans:**

- 1)  $\because$  **Kernel** 不知道有 **User Thread**, 只知道有 **P<sub>A</sub>** 與 **P<sub>B</sub>** 兩個 **Process**。  $\therefore$  **P<sub>A</sub>** 與 **P<sub>B</sub>** 各分到 **50%** 的 **CPU Time**。
- 2)  $\because$  有 **5** 條 **Thread** 欲分配, 每條可分到 **20%** 的 **CPU Time**。  $\therefore$  **P<sub>A</sub>** 分到  $3 \times 20\% = 60\%$  的 **CPU Time**; **P<sub>B</sub>** 分到  $2 \times 20\% = 40\%$  的 **CPU Time**。





## ■ Thread Pools (執行緒池)

- ✚ 每當O.S.收到一個要求時，就會產生一個個別的Thread來服務此要求。
- ✚ Multithread可以產生很多的Thread，但是可以產生多少呢？
  - ✚ 產生Thread需要花費時間
  - ✚ 如果無限制地生產，可能會耗盡系統的資源
- ✚ 有一個可能的解決方法就是Thread Pool的觀念。
- ✚ **Def:**
  - ✚ 一個Process開始執行時，產生了一些Thread，並將這些Thread放到一個Pool中以等待工作。
  - ✚ 當有一個工作要求產生時，就從這個Pool中喚醒一個Thread給要求者來執行其所需的工作。
  - ✚ 當工作完成後，Thread就回到Pool中等待其它的工作。
  - ✚ 如果一個工作要求產生，而Pool中沒有Thread可以執行時，這個工作就要等待直到有為止。







## ❁ Thread Pool的好處是：

- ❁ 對於一個服務要求而言，使用**現存的Thread**比等待產生一個**Thread**較來得快。
- ❁ 執行緒池限制了任何時間點上**Thread的個數**，不會讓系統的資源耗盡 (**Pool中沒有空閒的Thread時，不會再產生新的Thread**)。這對於**有資源上的限制**之系統非常重要。

## ❁ Thread Pool中的Thread個數可以根據：

- ❁ **CPU的個數**
- ❁ **實體記憶體大小**
- ❁ **預期客戶要求的個數**

